NSL: uma linguagem para o desenvolvimento e simulação de redes neuronais

Armando J. Pinho

Resumo – Neste artigo descrevemos o ambiente de simulação de redes neuronais artificiais que temos vindo a desenvolver, e também a utilizar, com vista à obtenção de resultados experimentais envolvendo este paradigma computacional. Este ambiente é baseado no NSL (*Neuro-simulation language*), que é uma linguagem interpretada destinada ao desenvolvimento e simulação de redes neuronais.

Abstract – In this paper we describe the simulation environment for artificial neural networks that we have developed, and also have used, with the aim of obtaining experimental results involving this computational paradigm. This environment is based on the NSL (*Neuro-simulation language*), which is an interpreted language for the development and simulation of neural networks.

I. INTRODUÇÃO

Associada à actividade de investigação, surge, com alguma frequência, a necessidade de simular modelos. São três, tipicamente, as vias que podem ser seguidas:

1. Se estiver disponível e for apropriado, pode-se utilizar *software* oferecido (ou vendido) por outros;
2. Desenvolve-se *software* mais ou menos *ad hoc* para resolver esse problema em particular;
3. Desenvolve-se um simulador relativamente flexível, não só com a finalidade de resolver o problema em causa, mas também permitir a abordagem de outros problemas similares que possam surgir no futuro.

A primeira das opções é, em geral, conveniente em estudos preliminares ou de reduzida dimensão. Contudo, quando começam a ser exigidas funcionalidades mais complexas, é fácil ser-se confrontado com limitações que gostaríamos de ultrapassar rapidamente. Isso nem sempre é possível quando se utiliza *software* de simulação desenvolvido por terceiros.

A segunda hipótese é seguramente a mais familiar para grande parte dos investigadores. Tipicamente, e no respeito ao desenvolvimento de *software*, é utilizado o mínimo esforço necessário para atingir os objectivos desejados. É óbvio que, do ponto de vista do investigador, o desenvolvimento de *software* é geralmente um meio e não um fim. No entanto, este carácter *ad hoc* é muitas vezes sinónimo de futuras dificuldades em reutilizar, estender ou adaptar esse *software*.

A terceira hipótese pressupõe a adição de um pouco mais de esforço no desenvolvimento do *software* de simulação. Este acréscimo é requerido especialmente na parte de planeamento e estruturação. É fundamental tentar prever o melhor possível futuras necessidades, para que o simula-

dor possa ser modificado ou estendido para as acomodar. Contudo, este esforço inicial é tipicamente compensado ao longo do período de utilização do simulador, devido à sua flexibilidade. Esta foi a opção que seguimos.

Neste artigo descrevemos, resumidamente, o ambiente de simulação de modelos neuronais que desenvolvemos e temos vindo a utilizar para a obtenção de resultados experimentais envolvendo este paradigma computacional. Esse ambiente é baseado numa linguagem interpretada, o NSL, que aqui descrevemos.

II. A LINGUAGEM NSL

NSL (*Neuro-simulation language*) é uma linguagem interpretada destinada ao desenvolvimento e simulação de redes neuronais. Contudo, dadas as suas características, pode ser usada como qualquer outra linguagem de programação geral. A sua estrutura foi fortemente inspirada pela linguagem de descrição gráfica POSTSCRIPT¹. A notação é em estilo *Postfix* (isto é, os operandos são sempre especificados antes do respectivo operador), associada à utilização de um *stack* de operação para a manipulação dos dados.

Até à presente forma, a linguagem NSL sofreu várias metamorfoses. O seu início remonta a 1988 com a implementação de um pequeno simulador do algoritmo de treino da retropropagação do erro [1]-[6]. Este simulador foi sendo melhorado através do aumento do número de opções e da sua flexibilidade, tendo atingido um estado de razoável desenvolvimento no início de 1991 aquando da sua utilização para a realização do trabalho da tese de mestrado do autor [7]. No Verão desse ano começou o desenvolvimento da linguagem StOP (*Stacked Object Programming*) que iria servir de interface de comando para um simulador de redes neuronais. Finalmente, foi feita a junção entre o interpretador de StOP e o núcleo do simulador, os quais até então tinham existido como dois programas separados comunicando através de FIFO's de UNIX, resultando na linguagem NSL que aqui descrevemos.

A. O interpretador

Vamos iniciar a apresentação da linguagem NSL utilizando alguns exemplos simples. Uma vez que a linguagem é interpretada, temos a possibilidade de interagir directamente com o interpretador, dando-lhe instruções e analisando o seu comportamento passo a passo. Sempre que o interpretador está pronto a receber comandos deverá apresentar um *prompt* (que, nos nossos exemplos, aparece como sendo "nsl>"). Portanto, após a inicialização, o interpretador deverá apresentar

¹ POSTSCRIPT é uma marca registada pela Adobe Systems Incorporated.

```
ns1>
```

Como já referimos, as operações são realizadas tendo como suporte um *stack*. Nele são colocados e retirados objectos de vários tipos, de forma a serem executadas as operações desejadas. Os primeiros objectos com que vamos operar são dos tipos *integer* e *real*. O tipo *integer* permite a manipulação de números inteiros com sinal, enquanto que o tipo *real* é utilizado para números representados em vírgula flutuante. Vamos ver o primeiro exemplo:

```
ns1> 234 23 sub
ns1>
```

Primeiro, os inteiros 234 e 23 são colocados no *stack* de operação. Em seguida, é realizada uma operação de subtração através do operador *sub*. Este operador requer dois operandos, os quais retira do *stack*. Após a operação ser realizada, o resultado, ou seja, o valor $211 = 234 - 23$, é depositado no *stack*. Este pequeno exemplo resume a forma geral de funcionamento da linguagem: os operandos são retirados do *stack* e os resultados são lá depositados, após as operações serem realizadas. Isto permite-nos, facilmente, encadear operações. Supondo que o *stack* contém o resultado da operação anterior, então

```
ns1> 3 mul 2 div 4 add
ns1>
```

origina a multiplicação de 211 (valor que estava no *stack*) por 3, seguida da divisão por 2, e ainda da adição do valor 4. Portanto, o *stack* deverá agora conter o objecto *real* "320.5".

Nos exemplos que temos dado até agora o *stack* tem permanecido invisível, tendo-nos limitado a imaginar o seu conteúdo. Contudo, podemos analisar o seu estado, sempre que desejarmos, usando o operador *stack*².

```
ns1> stack
320.5
ns1> pop stack
Empty stack
ns1> 1 2 3 4 5 stack
1 2 3 4 5
ns1> exch stack
1 2 3 5 4
ns1> clear stack
Empty stack
ns1>
```

Entretanto, foram introduzidos mais três operadores, os quais permitem manipulações de objectos no *stack* de operação: *pop*, *exch* e *clear*. O primeiro remove um elemento do topo do *stack*. O segundo comuta a posição relativa dos dois elementos do topo do *stack*. Finalmente, o terceiro remove todos os elementos do *stack*. Podemos ver também através deste exemplo que não é necessário mudar de linha para a introdução de um novo comando, isto é,

```
ns1> 1 2 add stack
3
```

²Note-se que, salvo indicação em contrário, as linhas que não começam com o *prompt* são escritas pelo interpretador.

```
ns1>
```

e

```
ns1> 1
ns1> 2
ns1> add
ns1> stack
3
ns1>
```

têm exactamente o mesmo efeito.

Vamos agora introduzir o objecto de tipo *string*. Todo o texto escrito entre parêntesis curvos (" " e " ") constitui um objecto do tipo *string*.

```
ns1> (hello world!)
ns1> (aeiou (uoiea) aei\ (ou)
ns1>
```

O segundo exemplo mostra também a utilização de parêntesis como fazendo parte do texto. Estes podem ser utilizados livremente, desde que sejam emparelhados ou então precedidos pelo carácter especial "\". Note-se também que no segundo exemplo temos um único objecto *string*, uma vez não existe recorrência na sua especificação. Um objecto *string* pode ser escrito no écran através do operador *print*.

```
ns1> (linha2\n) (linha1\n)
ns1> print print
linha1
linha2
ns1>
```

Este exemplo mostra o uso da sequência especial "\n" que indica mudança de linha. Aliás, outras sequências especiais comuns à linguagem C são também suportadas, tais como "\t" ou "\b". Mais uma vez vemos também a forma de funcionamento dos *stacks*, ou seja, o último objecto a entrar é o primeiro a sair. Daí que "linha1" seja escrito em primeiro lugar.

B. Agrupamento de objectos em arrays

Os objectos podem ser armazenados em objectos do tipo *array*. Um objecto do tipo *array* pode conter qualquer número de objectos de qualquer tipo. Existem duas formas de construir objectos do tipo *array*. Uma delas utiliza parêntesis rectos ("[" e "]") para delimitar os objectos a agrupar:

```
ns1> [24 44 6.89 (teste)]
ns1>
```

O *array* formado na operação anterior contém quatro objectos: dois do tipo *integer*, um do tipo *real* e um do tipo *string*. Uma vez que um *array* pode conter qualquer tipo de objecto, então podemos ter objectos do tipo *array* dentro de um outro objecto do tipo *array*:

```
ns1> [[1 2] [(a)]]
ns1>
```

Neste exemplo foi formado um objecto do tipo *array* constituído por outros dois objectos desse mesmo tipo que,

por sua vez, contém objectos do tipo `integer` e `string`.

A outra forma de gerar um objecto do tipo `array` é através do operador `array`:

```
nsl> 4 array stack
[-null- -null- -null- -null-]
nsl>
```

Este operador requer um operando do tipo `integer` indicando a dimensão pretendida para o `array`. Neste caso, o `array` é inicializado com objectos do tipo `null`, cuja única utilidade é o preenchimento de espaço.

Os principais operadores para a manipulação de objectos do tipo `array` são `get` e `put`. O primeiro extrai cópias de objectos armazenados em `arrays`. O segundo coloca objectos dentro do `array`. Os objectos de um `array` com n elementos são referenciados pelos inteiros 0 a $n - 1$:

```
nsl> clear [10 20 [1 2]]
nsl> stack
[10 20 [1 2]]
nsl> 2 get stack
[1 2]
nsl> 1 get stack
2
nsl>
```

Como vimos, o operador `get` requer dois operandos: um objecto do tipo `array` e um do tipo `integer`. Estes operandos são retirados do `stack`, pelo que ficamos sem hipótese de consultar de novo o `array`. Como ainda não falámos de variáveis, vamos socorrer-nos do operador `dup`, cuja função é a duplicação do elemento do topo do `stack`³. Vamos então ver um novo exemplo:

```
nsl> clear [10 20 30] dup stack
[10 20 30] [10 20 30]
nsl> 0 [1 2] put stack
[[1 2] 20 30]
nsl>
```

À partida, os objectos do tipo `array` têm uma dimensão fixa, que é dada ou pelo número de objectos existentes entre os parêntesis rectos ou pelo inteiro dado como argumento ao operador `array`. Contudo, essa situação pode ser alterada através do operador `cvdyn`. O seu efeito é a conversão do `array` para o modo dinâmico, permitindo, dessa forma, aumentar indefinidamente o seu tamanho⁴. Após a passagem ao modo dinâmico, todas as tentativas de `put` com índice maior ou igual à dimensão do `array` resultam no incremento da sua dimensão, sendo o novo elemento colocado no fim do `array`.

```
nsl> clear 0 array stack
[ ]
nsl> cvdyn stack
```

³O resultado obtido através da duplicação (`dup`) ou cópia (`copy`) de objectos é dependente do tipo de objecto em causa. Existem objectos simples (por exemplo, os de tipo `integer` e `real`) e objectos compostos (como é o caso dos de tipo `array` e `string`). A cópia ou duplicação de objectos compostos origina objectos que partilham os mesmos dados. Ou seja, se fizermos uma alteração num deles todas as suas cópias serão afectadas.

⁴Note-se que não existe a possibilidade de reduzir a sua dimensão.

```
[ ->]
nsl> dup 5 (aaa) put stack
[(aaa) ->]
nsl>
```

No exemplo anterior podemos ver que, embora o índice (5) exceda a dimensão do `array`, a operação é bem sucedida, tendo o objecto (`aaa`) sido colocado no fim do `array`⁵. Note-se que se a mesma operação fosse tentada num `array` estático resultaria num erro. Repare-se também no símbolo “->”, indicativo de que o objecto é dinâmico. Um `array` pode, em qualquer altura, ser convertido de volta para o modo estático, através do operador `cvstatic`.

C. Variáveis

O processamento de variáveis na linguagem NSL é feito através de dicionários. Um dicionário associa nomes a objectos, pelo que vamos passar a usar dois novos tipos de objectos: `dict` e `name`. Um objecto do tipo `name` é formado por um conjunto de caracteres não especiais e não susceptíveis de serem interpretados como um número. Um nome pode ser executável ou literal⁶. Sempre que um objecto do tipo `name` é apresentado ao interpretador é tomada uma de duas acções: se for literal, o objecto é simplesmente colocado no `stack`; se for executável, o interpretador tenta descobrir que objecto está associado a esse nome para que possa tomar uma acção. Vejamos o seguinte exemplo:

```
nsl> 12 4 add
nsl>
```

O interpretador encontra um objecto do tipo `integer` (o número 12), o qual coloca no `stack`. O mesmo procedimento é efectuado para o 4. Em seguida é encontrado um objecto do tipo `name` executável, cujo valor é “add”, pelo que é feita uma procura do objecto que se encontra associado ao nome “add”. Nessa procura é encontrado um objecto do tipo `operator` que ao ser executado origina o efeito final esperado.

A procura dos objectos associados aos nomes é feita em dicionários. À partida, existem dois dicionários activos e pré-definidos: `systemdict` e `userdict`. O dicionário de sistema, `systemdict`, contém todas as associações nome / operador, assim como alguns outros valores pré-definidos. O dicionário de utilizador, `userdict`, é principalmente destinado ao uso pelos programas criados pelo utilizador. Ambos são passíveis de alteração pelo utilizador, embora seja o `userdict` o mais indicado para esse fim.

Para que a procura de um nome seja efectuada automaticamente num dado dicionário, este tem que estar colocado no `stack` de dicionários. Este `stack` nada tem a ver com o de operação, só podendo conter objectos do tipo `dict`. Quando o interpretador é inicializado, existem somente dois dicionários no `stack` de dicionários: o `systemdict` no fun-

⁵Que, neste caso, coincide com o início, uma vez que o `array` estava vazio.

⁶Aliás, como qualquer outro objecto. No entanto, alguns dos objectos que já vimos possuem o mesmo comportamento tanto quando têm o atributo `literal` como quando são executáveis. Isto é o caso dos objectos de tipo `integer` e `real`. Por outro lado, objectos como os do tipo `string` ou `array` são sensíveis a este atributo.

do e o `userdict` no topo. Os nomes são procurados do topo para o fundo do *stack*. Num dicionário, um nome só pode corresponder a um objecto, isto é, não podem existir múltiplas definições. Contudo, o mesmo nome pode ter significados diferentes em dicionários diferentes. O valor que é utilizado é sempre o que se encontra mais acima no *stack* de dicionários. Vejamos um exemplo:

```
nsl> 10 dict stack
dict(0/10)
nsl> begin
```

Na primeira linha é criado um objecto do tipo `dict` com capacidade para 10 associações nome / objecto. Através da indicação dada pelo operador `stack`, vemos que o número de entradas usadas é 0 e que a sua capacidade máxima é 10. Em seguida, é utilizado um novo operador, `begin`. A sua função é colocar um dicionário no topo do *stack* de dicionários. O dicionário que está no topo do *stack* de dicionários designa-se “dicionário corrente”, do qual podemos obter uma cópia usando o operador `currentdict`. Vejamos agora como podemos associar nomes a objectos, no contexto do dicionário corrente:

```
nsl> /add 3.14 def
nsl> 1 2 add stack
1 2 3.14
nsl>
```

Como podemos ver, parece que o operador de adição, `add`, não funcionou, o que é verdade. Analisemos o que se passou. Na primeira linha do exemplo, associámos o real 3.14 ao nome `add` (para que o nome não seja considerado executável tem que ser escrito com o carácter “/” em prefixo). Isto quer dizer que a definição mais acima no *stack* de dicionários, relativa ao nome “`add`”, passou a ser essa e não a que relaciona o operador adição com esse nome. Repare-se que essa definição se encontra no dicionário `systemdict`, que está no fundo do *stack* de dicionários. Portanto, na segunda linha, o efeito de executar o nome `add` é a colocação no *stack* de operação do objecto real 3.14. Avancemos um pouco mais com o exemplo:

```
nsl> clear currentdict stack
dict(1/10)
nsl> clear
nsl> end
nsl> 1 2 add stack
3
nsl>
```

Após a colocação de uma cópia do dicionário corrente no *stack*, para mostrarmos que agora é indicada a utilização da entrada usada no exemplo anterior, usámos o operador `end` para retirar um dicionário do topo do *stack* de dicionários. O seu efeito é, por conseguinte, semelhante ao que o operador `pop` tem sobre o *stack* de operação. Após isso, pudemos executar, normalmente, a operação de adição. Embora não seja uma prática muito recomendada, os exemplos anteriores servem para ilustrar a grande flexibilidade da linguagem NSL.

Os dicionários também podem ser acedidos directamente através dos operadores `get` e `put`, de uma forma semelhante ao que já vimos para os objectos do tipo `array`. Contudo, neste caso as chaves de acesso são nomes em vez de valores inteiros.

```
nsl> /dc 5 dict def
nsl> dc /key1 (bla) put
nsl> dc /number 24 put
nsl> dc /number get stack
24
nsl> dc /key1 get stack
24 (bla)
nsl>
```

Na primeira linha criámos um dicionário e atribuímos-lhe o nome `dc`, no contexto do dicionário corrente. Em seguida, introduzimos no dicionário as associações `key1 / (bla)` e `number / 24`. Por fim, acedemos aos objectos anteriormente guardados através dos seus respectivos nomes.

Os dicionários são criados com uma capacidade limitada. Contudo, de forma semelhante aos objectos do tipo `array`, podem ser tornados dinâmicos ou estáticos usando os operadores `cvdyn` e `cvstatic`, respectivamente.

D. Matrizes

Já vimos como podemos agrupar objectos em `arrays`. Se desejarmos uma matriz de objectos, então podemos criar um `array` de `arrays` tal como, por exemplo,

```
nsl> [ [1 2 3] [4 5 6] [7 8 9] ]
nsl>
```

onde é criada uma matriz de três por três elementos. O tipo `array` é, efectivamente, muito poderoso e flexível. Contudo, se pretendermos efectuar cálculo intensivo sobre matrizes constituídas desta forma, rapidamente somos confrontados com as limitações de velocidade inerentes a uma linguagem interpretada. Por outro lado, grandes matrizes necessitam de grandes quantidades de memória, uma vez que cada elemento da matriz (construída de acordo com o exemplo dado) é um objecto genérico. Para contornar estes problemas criámos o tipo `matrix`, cujas principais funções são o armazenamento e manipulação eficiente de objectos numéricos.

```
nsl> /mat 3 5 4 matrix def
nsl> mat stack
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
nsl>
```

Neste exemplo vimos como criar uma matriz de três linhas por cinco colunas. O terceiro argumento do operador `matrix` indica, em *bytes*, o espaço de memória usado para armazenar cada elemento. Actualmente, são permitidos os tamanhos 1, 2 e 4 para inteiros, e o tamanho 8 para reais⁷. Continuemos com o exemplo:

```
nsl> mat 0 1 put
```

⁷Isto corresponde, respectivamente, aos tipos `char`, `short`, `int` e `double` da linguagem C.

```
nsl> mat 8 5 put
nsl> stack
1 0 0 0 0
0 0 0 5 0
0 0 0 0 0
nsl>
```

O acesso aos elementos da matriz é feito através de um índice sequencial, tal como podemos ver no exemplo dado com o operador `put`. O operador `get` actua da mesma forma.

```
nsl> mat /Fill [1 2 3] do
nsl> stack pop
1 2 3 1 2
3 1 2 3 1
2 3 1 2 3
nsl> mat /Rotate [1] do
nsl> stack
2 1 3
1 3 2
3 2 1
2 1 3
1 3 2
nsl>
```

Neste exemplo introduzimos o operador `do`, cuja função é ordenar acções sobre um objecto. Neste caso vimos duas dessas acções, `Fill` e `Rotate`, que são permitidas sobre objectos do tipo `matrix`⁸. A primeira acção origina que a matriz seja preenchida com a sequência dada como argumento. Esta é, aliás, uma das possíveis formas de inicializar uma matriz. A segunda acção produziu a rotação da matriz de noventa graus no sentido directo. A rotação que é efectuada é igual a noventa graus vezes o valor do argumento módulo quatro.

A introdução do tipo `matrix` na linguagem tornou-se fundamental para a manipulação eficiente de imagens. Entre outras particularidades, é possível, por exemplo, ler e escrever matrizes em ficheiros através de uma única instrução.

E. Procedimentos

Os procedimentos em NSL são obtidos à custa de um objecto do tipo `array`. A única diferença entre um `array` procedimento e um `array` "normal" é o seu atributo de execução: os procedimentos são objectos do tipo `array` executáveis, enquanto que os outros são literais.

Existem duas formas de construir procedimentos. Uma delas é através da construção de um `array` usando uma das formas que vimos, convertendo-o em seguida em executável por meio do operador `cvx`. A outra forma, que é a mais natural, utiliza os caracteres especiais "{" e "":

```
nsl> { add 2 div }
nsl>
```

Este exemplo simples mostra a criação de um procedimento cuja execução origina a adição dos dois valores existentes no topo do `stack`, seguida da divisão por 2. Vejamos um ou-

tro exemplo, onde construímos um procedimento que converte polegadas em centímetros:

```
nsl> /inch2cm { 2.54 mul } def
nsl> 4.5 inch2cm stack
11.43
nsl>
```

F. Operadores de controlo

O fluxo normal de execução do interpretador de NSL pode ser alterado por um dos vários operadores de controlo disponíveis. Começemos por introduzir o tipo de objecto `boolean`. Este objecto só possui dois valores possíveis: verdadeiro (`true`) ou falso (`false`). Existem diversos operadores que originam objectos deste tipo. Vejamos um pequeno exemplo:

```
nsl> /testexec {
nsl>   xcheck {
nsl>     (Executable\n) print
nsl>   } {
nsl>     (Literal\n) print
nsl>   } ifelse
nsl> } def
nsl> [ 1 2 ] testexec
Literal
nsl> [ 1 2 ] cvx testexec
Executable
nsl>
```

Começámos por construir um procedimento com o nome `testexec` cuja função é retirar um objecto do `stack`, analisar o seu atributo de execução (através do operador `xcheck`) e escrever a palavra "Executable" ou "Literal" conforme o caso. A decisão é tomada por meio do operador de controlo `ifelse`. Este operador requer três operandos: um objecto do tipo `boolean` e dois procedimentos. Se o objecto `boolean` tiver o valor `true`, então é executado o primeiro procedimento, caso contrário é executado o segundo. Neste exemplo, o valor `boolean` é fornecido pelo operador `xcheck`, que retorna `true` se o seu operando for executável ou `false` se for literal. Vejamos dois outros operadores de controlo:

```
nsl> 0 1 5 { } for
nsl> stack
0 1 2 3 4 5
nsl> 4 { pop } repeat
nsl> stack
0 1
nsl>
```

O operador `for` executa ciclos controlados por uma variável interna, à qual não temos acesso. Essa variável é inicializada pelo primeiro operando (0, neste caso) e, em cada ciclo, sofre uma variação igual ao valor do segundo operando (1, no exemplo). O fim dos ciclos é controlado pelo valor limite indicado pelo terceiro operando (que é 5, neste caso). Em cada ciclo é depositado no `stack` de operação o valor da variável interna, e o quarto operando, que deve ser um procedimento, é executado. Uma vez que no exemplo apresentado o procedimento era vazio, então o único efeito

⁸Existem outras, mas não as iremos descrever aqui.

foi a colocação dos valores da variável de ciclo no *stack*.

O operador *repeat* executa um número de ciclos igual ao valor do seu primeiro operando. Em cada ciclo, o segundo operando, que deve ser um procedimento, é executado.

Vamos ainda apresentar um outro operador de controlo, *forall*, cuja função é aplicar, sistematicamente, um procedimento a cada elemento constituente de um objecto do tipo *array*, *dict*, *string* ou *matrix*:

```
nsl> [1 3 7] { 2 mul } forall
nsl> stack
2 6 14
nsl>
```

No exemplo anterior podemos ver que o procedimento "{ 2 mul }", ou seja, a multiplicação por dois do elemento no *stack*, foi aplicada à totalidade do *array* através de um único operador.

G. Redes neuronais, unidades e ligações

Para a manipulação de redes neuronais vamos introduzir mais 3 novos tipos de objecto: *network*, *unit* e *link*. Uma rede neuronal é constituída por um conjunto de elementos processadores (unidades) interligados de acordo com a topologia pretendida. A linguagem NSL permite a criação de redes neuronais através do operador *network*, o qual retira do *stack* dois operandos: um objecto do tipo *integer*, indicando o número de unidades da rede, e um objecto do tipo *name* que define o tipo de rede.

```
nsl> 20 /Default network
nsl>
```

As redes do tipo *Default* contêm estruturas de dados que, em princípio, são suficientes para a implementação de uma grande variedade de arquitecturas e algoritmos de aprendizagem. Contudo, por motivos de eficiência, podemos desejar implementações diferentes. Note-se que a linguagem NSL é interpretada, pelo que sofre de limitações quanto à sua velocidade de execução. Isso pode ser resolvido através da adição de novos módulos, como é o caso do que implementa o algoritmo de aprendizagem da retropropagação do erro.

Os objectos do tipo *network* são manipulados, fundamentalmente, através de três operadores: *get*, *put* e *do*. O operador *get* permite a leitura de parâmetros internos dos objectos *network*. Por outro lado, o operador *put* destina-se a modificar alguns desses parâmetros. Por fim, o operador *do* é dedicado à transmissão de ordens que desencadeiam acções associadas ao objecto. Vejamos o seguinte exemplo:

```
nsl> /net 5 /Default network def
nsl> net /Units get stack
5
nsl> net /RunMode get stack
5 /Undefined
nsl> clear
nsl> net /ExecList get stack
[-unit- -unit- -unit- -unit- -unit-]
nsl> net /RunMode /Sequential put
nsl> net /Save [(test.net)] do
```

```
nsl>
```

Primeiro, foi construída uma rede neuronal com cinco unidades. Em seguida, foi mostrada a forma geral de acesso para leitura dos parâmetros internos do objecto *network*, tendo sido dados como exemplos a pesquisa do número de unidades que constituem a rede (*Units*), o modo de funcionamento (*RunMode*), e a lista de unidades sujeitas a actualização (*ExecList*). Actualmente, são suportados dois modos de funcionamento: o modo sequencial (*Sequential*) que, em cada ciclo de relógio, provoca a actualização do estado de todas as unidades que aparecem na lista de execução (*ExecList*), segundo a sua ordem nessa lista; o modo assíncrono (*Asynchronous*) que, em cada ciclo de relógio, produz a actualização do estado de uma fracção (*AsyncFraction*) das unidades existentes na lista de execução, sendo a ordem de escolha aleatória. As duas últimas linhas do exemplo mostram, por um lado, a alteração de um dos parâmetros internos da rede (o seu modo de funcionamento) e, por outro, a utilização do operador *do*. Este operador espera três operandos: o objecto sobre o qual vai exercer a acção, neste caso um objecto do tipo *network*, um nome que identifica a acção, e um *array* de argumentos. No caso do exemplo dado, a acção é a gravação da rede num ficheiro com o nome "test.net".

Os objectos do tipo *unit* (ou seja, as unidades da rede) não podem ser criados isoladamente, não havendo por isso um operador "unit", como seria de esperar. As unidades são criadas automaticamente através do operador *network*, como vimos atrás. A forma de obtermos qualquer unidade ou conjunto de unidades de uma rede neuronal é a seguinte. Para isso, vamos supor que a rede se encontra criada e com o nome "net".

```
nsl> net 3 get stack
-unit-
nsl> net [1 2 5] get stack
-unit- [-unit- -unit- -unit-]
nsl>
```

Na primeira linha é obtida a unidade com o índice número três⁹. Na segunda linha de comando é solicitado o conjunto constituído pelas unidades 1, 2 e 5. O resultado é um *array* com essas unidades.

Os objectos do tipo *unit* são manipulados de uma forma semelhante aos do tipo *network*. Vejamos um exemplo, recordando que o operador de controlo *forall* aplica o procedimento a cada um dos elementos do *array*:

```
nsl> stack
-unit- [-unit- -unit- -unit-]
nsl> {/Index get} forall stack
-unit- 1 2 5
nsl> 3 { pop } repeat
nsl> /Type get stack
/Undefined
nsl>
```

⁹As unidades de uma rede neuronal são numeradas de 0 até ao valor do parâmetro "Units". Isto implica que numa rede declarada como tendo *n* unidades existem, efectivamente, *n* + 1. A unidade com o índice zero é criada automaticamente e é destinada ao uso como unidade auxiliar para os pesos de polarização, tendo por isso a sua saída fixa e igual a 1.

O parâmetro *Index* contém o índice correspondente à unidade em causa. Por outro lado, cada unidade é caracterizada por um tipo (*Type*) que define a função de transferência dessa unidade. Alguns dos tipos suportados são:

Threshold

Tipo usado pela unidade cujo índice é 0.

Input

Unidade cuja única função é a aceitação de valores exteriores.

Linear

Unidade com uma função de transferência linear, isto é, a saída é igual à entrada.

Hard

Unidade cuja saída é 0 ou 1 conforme a entrada é menor ou maior que zero, respectivamente.

Sigmoid

Unidade com uma não linearidade do tipo sigmóide, com valores de saída entre 0 e 1.

Sigmoidsym

Unidade com uma não linearidade do tipo sigmóide, com valores de saída entre -1 e 1.

Vejamos um outro exemplo, onde são feitas algumas operações com unidades:

```
nsl> clear /unit1 net 1 get def
nsl> unit1 /Type /Sigmoid put
nsl> unit1 /Output get stack
0
nsl> unit1 /Input [3.18] do
nsl> unit1 /Output get stack
0 0.960075
nsl> clear
nsl> unit1 /Run [] do
nsl> unit1 /Input get stack
0
nsl> unit1 /Output get stack
0 0.5
nsl>
```

Após a atribuição de um nome à unidade cujo índice é 1, fizemos a sua configuração de modo a ter uma função de transferência do tipo sigmóide. Em seguida, consultámos o seu valor de saída, que se encontra armazenado no parâmetro *Output*. O passo seguinte foi forçar o valor 3.18 na entrada da unidade (*Input*) e analisar o resultado de saída. Finalmente, utilizámos a acção *Run* para comunicar uma ordem de actualização à unidade. A acção *Run* é uma actualização completa da unidade: primeiro, é calculado o valor de entrada que, neste caso, é zero, uma vez que não existem ligações; em seguida, é calculado o valor de saída, através da respectiva função de transferência da unidade.

Vamos agora estabelecer ligações entre as unidades. Cada ligação é representada por um objecto do tipo *link*. Mais uma vez, a forma de acesso a este tipo de objectos é semelhante à que temos vindo a ver para as unidades e redes. Uma ligação entre duas unidades é conseguida através do operador *link*:

```
nsl> /unit1 net 1 get def
```

```
nsl> /unit2 net 2 get def
nsl> unit1 unit2 0.1 link stack
-link-
nsl>
```

Além da unidade fonte (neste caso "unit1") e da unidade destino ("unit2") o operador *link* requer também um valor numérico com o qual inicializa o peso da ligação. Os pesos podem ser acedidos através do parâmetro *Weight*. Podemos também obter os objectos do tipo *unit* que constituem a fonte e o destino da ligação, por meio dos parâmetros *From* e *To*, respectivamente. Aproveitando o estado do *stack* do exemplo anterior, eis mais algumas linhas de NSL:

```
nsl> stack
-link-
nsl> /link1 exch def
nsl> link1 /Weight get stack
0.1
nsl> link1 /Weight 1.25 put
nsl> link1 /To get stack
0.1 -unit-
nsl> /Index get stack
0.1 2
nsl>
```

Para terminar esta secção, vamos apresentar mais um exemplo que achamos importante, uma vez que demonstra a partilha de pesos.

```
nsl> /net 10 /Default network def
nsl> /unit1 net 1 get def
nsl> /unit2 net 2 get def
nsl> /unit3 net 3 get def
nsl> /linkA unit1 unit2 0 link def
nsl> /linkB unit1 unit3 0 link def
nsl> net /ShareWeights
nsl> [linkA linkB] do
nsl> linkA /Weight get stack
0
nsl> linkB /Weight 0.5 put
nsl> linkA /Weight get stack
0 0.5
nsl>
```

Neste exemplo, foi criada uma rede e efectuada uma ligação entre as unidades 1 e 2, e entre as unidades 1 e 3. Em seguida, foi executada a acção que declara partilha de pesos (*ShareWeights*), cujo efeito pode ser constatado pelo resto do exemplo. A lista de ligações (objectos do tipo *link*) que é fornecida como argumento à acção *ShareWeights* pode ter qualquer número de elementos.

III. CONCLUSÕES

Neste artigo apresentámos uma introdução à linguagem de simulação e desenvolvimento de redes neuronais, NSL. Esta linguagem comporta, actualmente, mais de 150 operadores que operam directamente 16 tipos de objectos diferentes, e ainda 8 tipos de forma indirecta. O interpretador de NSL encontra-se implementado em aproximadamente 26 000 linhas de linguagem C, tendo sido desenvolvido em

ambiente UNIX. Dada a facilidade oferecida pela linguagem na criação de novos comandos, já existem também algumas extensões desenvolvidas e escritas na própria linguagem, como vimos, aliás, quando referimos comandos como `LinkSets`.

Um dos principais problemas da presente implementação é a não existência de um colector de objectos obsoletos (*garbage collector*) ou, mais genericamente, de um gestor de memória. Isto faz com que a memória seja continuamente consumida. Para minorar este problema existe um destruidor explícito de objectos (o operador `destroy`). Embora este ponto seja uma lacuna importante na presente implementação nunca fomos confrontados com uma situação crítica de esgotamento de memória.

Existem várias hipóteses para o alargamento da linguagem. Um dos mais interessantes seria a introdução de um novo objecto do tipo `stack`. Entre outras vantagens este tipo de objecto poderia permitir a simulação de multi-processamento, através da manutenção de vários *stacks* de execução. Por outro lado, e à semelhança com o que se passa com o algoritmo de treino da retropropagação do erro, é bastante desejável que outros módulos sejam também acrescentados ao simulador.

REFERÊNCIAS

- [1] A. E. Bryson e Y.-C. Ho, *Applied optimal control*, Blaisdell, New York, 1969.
- [2] P. Werbos, *Beyond regression: new tools for prediction and analysis in the behavioral sciences*, PhD thesis, Harvard University, Cambridge, November de 1974.
- [3] Y. le Cun, "Une procedure d'apprentissage pour reseau a seuil asymetrique", em *Proc. Cognitiva'85, Paris, France*, June de 1985, pp. 599-604.
- [4] D. B. Parker, "Learning-logic", Relatório Técnico TR-47, MIT Cen. Computational Res. Economics Management Sci., Cambridge, 1985.
- [5] D. E. Rumelhart, G. E. Hinton, e R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, no. 9, pp. 533-536, October de 1986, Reprinted in [8].
- [6] D. E. Rumelhart e J. L. McClelland, Eds., *Parallel distributed processing — explorations in the microstructure of cognition: foundations*, vol. 1, MIT Press, Cambridge, MA, 1986.
- [7] A. J. Pinho, "Compressão de imagem por redes neuronais", Master's thesis, Instituto Superior Técnico, Lisbon, Portugal, May de 1991, (in portuguese).
- [8] J. A. Anderson e E. Rosenfeld, Eds., *Neurocomputing: Foundations of research*, MIT Press, Cambridge, 1988.